THE DEVELOPMENT OF A SEGMENTED MEMORY MANAGER FOR THE UNIX OPERATING SYSTEM WITH APPLICATIONS IN A MULTIPORTED MEMORY ENVIRONMENT.

James M. O'Dell

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

The Development of a Segmented Memory Manager for the UNIX Operating System with Applications in a Multiported Memory Environment

by

James M. O'Dell

September 1977

Thesis Advisor: G. L. Barksdale

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): The Development of a Segmented Memory Manager for the UNIX Operating System with Applications in a Multiported Memory Environment	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; September 1977
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): James M. O'Dell	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: September 1977
	13. NUMBER OF PAGES: 79
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Naval Postgraduate School, Monterey, California 93940	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

segmentation, multiported memory, real-time processing, UNIX, memory management, operating system, shared core, system protection

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis reports the development of a segmented memory manager for the UNIX operating system on a PDP-11/50 minicomputer. Considered in detail is the application of this memory manager to data segment boundary alignment and system protection in a multiported memory

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

configuration. Recommendations are provided for the integration of this operating system into the total Naval Postgraduate School Signal Processing and Display Laboratory Environment.

Approved for public release; distribution unlimited

The Development of a Segmented Memory Manager for the UNIX Operating System with Applications in a Multiported Memory Environment

by

James M. O'Dell
Lieutenant, United States Navy
B.S., United States Naval Academy, 1970

Submitted in partial fulfullment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1977

ABSTRACT

This thesis reports the development of a segmented memory manager for the UNIX operating system on a PDP-11/50 minicomputer. Considered in detail is the application of this memory manager to data segment boundary alignment and system protection in a multiported memory configuration. Recommendations are provided for the integration of this operating system into the total Naval Postgraduate School Signal Processing and Display Laboratory environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

The goal of the Naval Postgraduate School Signal Processing and Display Laboratory is to provide a facility for research and education in the field of computer science. The laboratory is built around a pair of Digital Equipment Corporation PDP-11/50 processors. One processor is primarily used to service multiuser program development activities. The other processor supports several graphics display devices and provides a dedicated environment for research and development in the areas of operating systems, computer graphics and signal processing. Figure 1 shows the present hardware configuration of the laboratory.

The UNIX operating system [7], a time-sharing system developed at Bell Laboratories, was chosen as the system best suited to support the goals stated above. UNIX's inherent file system flexibility and the availability of system source code written in the high level programming language, "C", provide an excellent environment for operating system development and research. The availability of a dedicated processor as a developmental tool further enhances this environment.

Due to the unique and demanding requirements placed on the display processor's operating system by graphics devices and signal processing equipment, it was determined that the standard version of UNIX in use (henceforth simply referred

to as UNIX) was unsatisfactory. This thesis proposes an extension to the UNIX system in the form of a new memory management scheme. The approach taken was to implement process segmentation with provision for data space allocation and alignment within a specified memory region. This approach was feasible and attractive because the PDP-11/50 is equipped with a memory management unit (MMU) that supports segmentation. Additionally, segmentation provides extra flexibility in resident process manipulation. UNIX allocates process space as a single contiguous unit in main memory. Segmentation allows management of process segments as separate entities. This provides the necessary tool for selective allocation of memory space to an individual segment.

This latter feature is particularly attractive in a system configured with multiported memory. As shown in Figure 1, the display processor has access to two dual-ported memory areas: one shared with a signal processing controller, the other shared with a slave processor for the refresh graphics display devices. Due to the real-time requirements of these devices, independent access to data in the shared region is necessary. Otherwise, the display processor will not be able to support multiprogramming. A segmented memory manager would provide the features necessary to support these real-time requirements.

Two modifications of the UNIX operating system were implemented and tested as a result of this thesis. One was a previously designed [4], but unimplemented, segmented

memory manager version (SUNIX). The other, a shared-segmented version (SSUNIX), was a further modification of SUNIX which implemented the alignment of a process's data space within a shared memory region. Real-time processing was supported by locking the process image in main memory once the shared region had been acquired.

Testing of SUNIX and SSUNIX included benchmark measurements and operation in the general multiuser environment. Performance was almost identical to that of UNIX. Additionally, SSUNIX was tested on the display processor and successfully fulfilled the objective of data space alignment with system protection. Amplification of these results is provided in Chapter V of this thesis. In addition, recommendations are included for applications and utilization of SUNIX and SSUNIX in the Signal Processing and Display Laboratory.

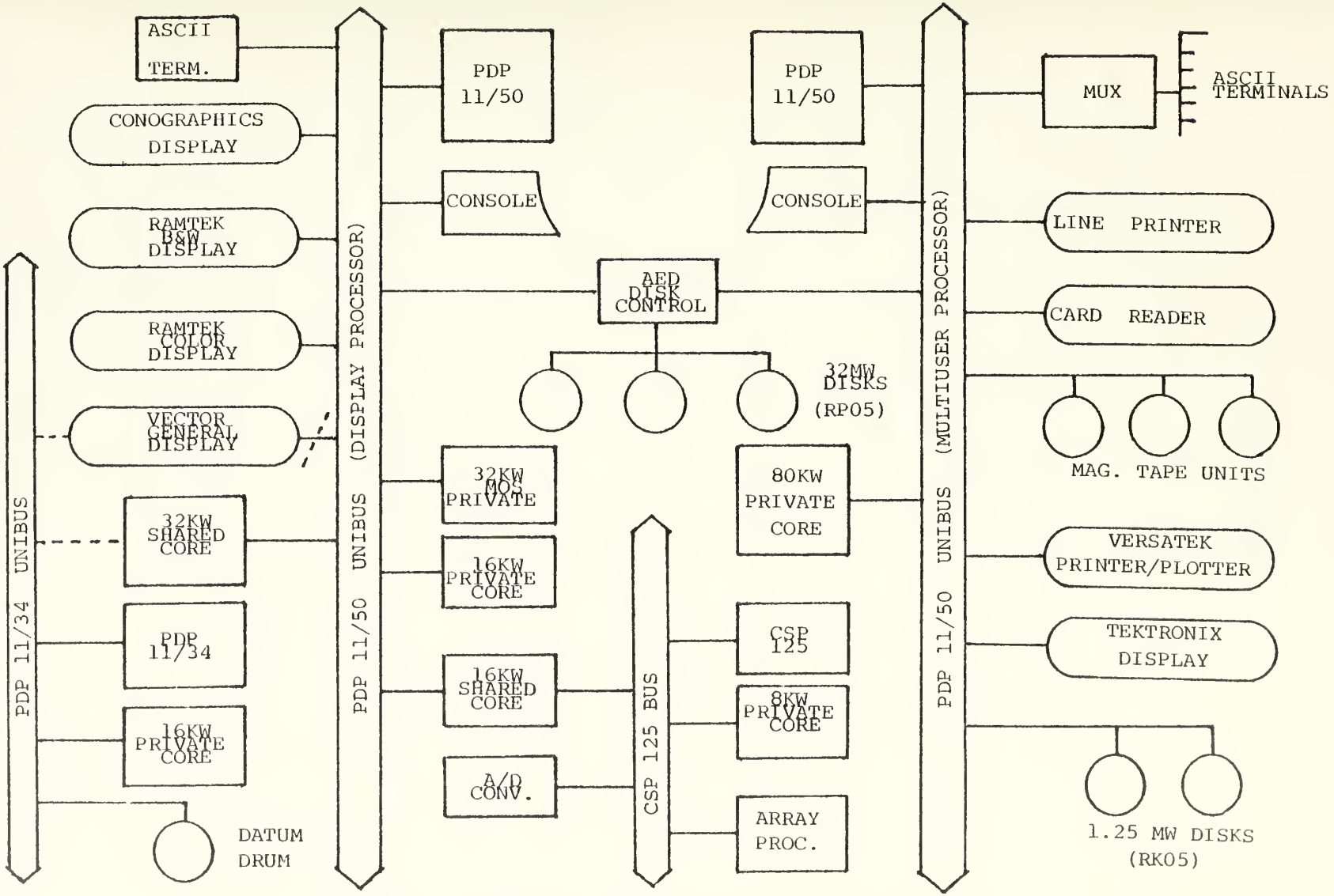

FIGURE 1. Signal Processing and Display Laboratory

II. BACKGROUND

A. PREVIOUS WORK

Three previous thesis projects which are closely associated with the development of SUNIX and SSUNIX are discussed below. It should be noted that a specific area of concern in all three papers is the development of system support for the special devices on the display processor.

1. Partitioned Segmented Memory Manager

In this thesis [4], Emery reports an investigation of the applicability of paging and segmentation to memory management in UNIX. Two memory managers were designed to support this investigation: a partitioned segmented version (PSUNIX), and a simpler segmented version (SUNIX). The primary consideration in the design effort was that the segments be separate and independently manageable in main memory. The segmentation chosen was the existing natural division into a process control block, a data segment (including non-shared text), and a User stack. Shared text segments were handled in much the same manner as in UNIX. SUNIX was designed to manage each complete segment independently. It was this design effort that provided a foundation for the development of SSUNIX. PSUNIX further broke down each of the segments into variable-size blocks.

Testing of PSUNIX was accomplished by using

benchmark measurements to analyze its performance in relation to UNIX. The experimental results clearly indicated that the performance of PSUNIX and UNIX were nearly identical over a wide range of available User memory space. Emery suggested that this approximate equality of pergormance indicated that the disadvantage of the increased amount of swapping in PSUNIX was offset by a reduction in the number of processes swapped. He attributed this to reduced external fragmentation with PSUNIX. However, due to the relative complexity of PSUNIX, Emery recommends that the simpler SUNIX be completed and implemented as a viable alternative.

2. Inter-Processor Graphics Support System

Visco [12] investigated a multiple processor system configuration to support a real-time, interactive graphics package. UNIX was redesigned to implement a real-time system call and a master-slave relationship between the display processor and a dedicated graphics processor. Specifically, he designed the system support routines necessary for integration of the PDP-11/50, PDP-11/34, and Vector General (VG) Display Processors. For testing purposes, inter-processor communications were simulated on the PDP-11/50, since the PDP-11/34 slave processor was not available for use.

The master-slave processor configuration was conceived as an answer to the problem of refresh graphics devices having a high level of direct memory access (DMA)

requests and a requirement for real-time system support. DMA is the transfer of data directly to or from memory by a peripheral device, without processor supervision. This configuration required shared memory and the system support software to align a process image in the shared region. Memory management hardware on the slave processor [2] permitted a 32 thousand (K) word address range. The system designed by Visco allocated the entire process image on a 32K word boundary. System protection was not achieved since other processes were allowed to overlap into this region.

Visco suggested an implementation of Emery's SUNIX as a method of allocating only the graphics process's data space to the shared region of memory. Since DMA by the slave processor, acting in behalf of the graphics device, was highly concentrated in data space, this memory allocation scheme would more efficiently utilize the limited amount of shared memory. SSUNIX, with the added features of generalized multiported memory applications and system protection, was developed as a result of this suggestion.

3. Loosely-Coupled Multiprocessor System

Visco's work was directed primarily toward the main processor in the proposed multiprocessor system for the laboratory. A thesis by Gray [5] addresses the development of an operating system for the slave processor. Although not directly related to the problem of memory management in UNIX, this paper is mentioned here to emphasize the general trend of thesis work done in support of the laboratory's

refresh graphics devices. Furthermore, recommendations in a later chapter of this paper specifically address the integration of SSUNIX into the multiprocessor environment.

B. GENERAL SYSTEM PROPOSAL

As indicated in the previous section, the specific system proposed by this thesis was designed primarily to perform data space alignment in a shared memory area for resident signal processing and refresh graphics processes. However, the actual modifications to UNIX were designed to offer more general applications. SUNIX was implemented as a general purpose operating system with segmented memory management. The differences between UNIX and SUNIX are transparent to the User. SSUNIX provides the ability to align a process's data space within any given area of User memory space. Location and size of this memory region, which was intended to correspond to shared memory assets, can be specified at system generation time. The limitations of UNIX in a system configured with devices which demand heavy DMA and real-time system support are discussed. The general concepts behind the design of SUNIX and SSUNIX are also presented.

1. Limitations of the Present System

a. Bus Allocation

As shown in Figure 1, all PDP-11/50 components and peripherals connect to and communicate with each other

over a high-speed, bidirectional, asynchronous bus known as the UNIBUS [3]. Devices gain access to the UNIBUS via an arbitration unit. This unit grants bus mastership based on a multilevel priority scheme. Each device on the UNIBUS is assigned a hard-wired priority which is recognized by the arbiter. A device is granted immediate mastership when a request is made and no device of equal or higher priority has control. DMA requests from peripheral devices have highest priority. The central processing unit (CPU) has the capability to control or release the bus by varying its own priority level. This makes it possible for devices to access main memory with almost no processor intervention.

Devices which require a large amount of DMA place heavy demands on the UNIBUS. A problem arises in a multiprogramming environment when a process is running at a high scheduling priority (as with real-time processes), and also requires high priority and heavy bus utilization for DMA. In this case, all other processes are essentially suspended since they cannot gain access to the UNIBUS. Refresh graphics devices require heavy bus utilization for processing the display list. In the case of the VG devices in the laboratory, the display list, located in the resident graphics process's data space, must be processed every one-fortieth of a second. It is thus necessary to lock the active display list in memory. If the list were not in memory at the time of a refresh cycle an inordinate delay would occur, causing the display to flicker or fade.

As noted previously, dual-ported memory was

conceived as a solution to these problems. With the peripheral processors each assigned a separate bus attached to memory ports opposite the main processor, interference during DMA could be virtually eliminated. The problem which remained was the allocation of a process's data space to one of the shared memory areas. Visco's design attempted to solve this problem specifically for VG processes. SSUNIX was designed to solve the problem for more general applications.

b. Memory Allocation Scheme

In UNIX, while the processor is executing in behalf of a process, its image must reside in main memory as a single contiguous unit. Unless swapping is necessary, the image remains in memory during the execution of other processes. When the CPU is executing a process, MMU registers are loaded so the process can access only its own image and, if applicable, a text segment shared with other processes.

The memory allocation scheme used in UNIX is based on a "first-fit" algorithm. That is, the process image is allocated space in the first free area in the system's free-list which can accomodate it. With this scheme, selective allocation and alignment of memory space in a particular region of memory is impossible. This is unsatisfactory in a system configured with multiported memory. As an example, the Computer Signal Processors Incorporated CSP 125 controller can access only that portion

of memory indicated in Figure 1. Any process that communicates with the CSP 125 must have its data space located in that area of memory.

2. General Features of the Proposed System

a. Segmentation

The initial decision that had to be made to solve the problems stated above was: what type of memory management modifications to UNIX were required to provide a general and efficient operating system which would support alignment of a resident process's data space in a given region of memory. The segmented memory manager designed by Emery (SUNIX) was chosen as a basis for the intended modifications. Segmentation had several advantages which made this choice attractive. Process images in UNIX consist of several logically independent segments. Although not managed separately, this natural division provides an appealing basis for segmentation. Modifications to the existing memory management routines to implement SUNIX were straightforward and relatively simple.

The segmentation chosen also proved to be an efficient way to handle dynamic changes in the size of User data and stack areas. In UNIX, when the User space grows beyond the available contiguous memory space, it is necessary to reestablish the entire process image in main memory. This is accomplished by copying the process image to a new free area of sufficient size. Since there is no hardware facility on the PDP-11/50 for a "block move" in

memory, this is a major source of memory management overhead. The cost of the copy operation is about 10 micro-seconds per word [4]. This figure is even worse if space is not readily available for the larger image and other processes have to swapped out to make room. In SUNIX the User's data and stack are managed independently. Growth of either area requires reestablishing only that segment in memory. Total overhead due to dynamic in-core expansion is, therefore, reduced.

b. Data Space Alignment

After deciding on SUNIX as the operating system which would best serve the purposes of this thesis, a method of positioning a process's data space at a particular location in main memory had to be developed. Detailed analysis of the memory management routines in both SUNIX and UNIX was necessary before proceeding. The method chosen, and implemented in SSUNIX, was a direct extension of SUNIX. No modification to the segmentation scheme was necessary, and only minor modifications to the other memory management routines were required. The general approach taken was to implement the necessary changes by adding several system routines to acquire, allocate, and release a shared memory asset. System calls were implemented to perform these functions for processes requiring shared memory.

Providing system protection was a secondary goal in the design of a memory manager for SSUNIX. The concept of system protection implies that inadvertent destruction of

vital process information, particularly the process control information used by the operating sytem, will not occur. Address range limitations of the peripheral processors prevent their manipulation of data in memory locations outside of the shared region. SSUNIX allows only the data space of processes communicating with these devices in this region. The combination of these two features provided the system protection desired.

III. THE UNIX OPERATING SYSTEM

Thus far in this thesis, the UNIX operating system has been discussed in general terms. This chapter provides a more detailed discussion of the concepts of system operation. Concepts in the design of the memory managers in UNIX, SUNIX and SSUNIX are also presented. This chapter provides the background necessary for understanding the modifications made to UNIX to implement SUNIX and SSUNIX.

A. CONCEPTS OF OPERATION

UNIX [7] is a terminal oriented, time-sharing operating system developed at Bell Laboratories for the Digital Equipment Corporation (DEC) PDP-11 family of minicomputers. Most of the source code for UNIX is written in "C" [9], a high level systems implementation language. The remainder of the source is written in "as" [10], a Bell Laboratories variant of PDP-11 assembly language.

Multi-tasking in UNIX is performed on a basic unit of work called the process. A process consists of system control information, executable instructions, and data. When the operating system is "bootstrapped" into memory at system initiation time, it "handcrafts" its first two processes. Process 0 is the master control process (the scheduler "loop") which executes until UNIX terminates.

Process 1 initializes the operating environment for other processes in such a manner that all subsequent processes are its descendents. All other processes are executions of program files from the UNIX file system.

Descendents of a process are created by execution of the "fork" system call. This system call replicates the calling process, creating a new (child) process that has a unique process number. The original process, called the parent, can continue execution or temporarily suspend itself until the child terminates. A child process may either continue execution of the same program or invoke a new program into execution. New processes are invoked by "exec", a system call that "overlays" the calling process with an executable file from the file system. Child processes may also spawn children of their own. When any child completes execution, it terminates by means of the "exit" system call. "Exit" notifies the parent of the child's termination.

The primary role of Process 1 is to create a child process for each of the terminals in the system. These processes open their assigned terminals, prompt for a user to log in, and then await a reply. Once a user has logged in, the child invokes a new program (using "exec") called the Shell. The function of the Shell is to interpret commands from the terminal and create children which accomplish the desired operation. When the user logs off, the Shell process for his terminal is terminated. Process 1, which is then notified of that child's demise, creates another child for the terminal. The new child reopens the

terminal and prompts for another log in.

From the User's point of view, the most important role of UNIX is to provide a file system [7]. There are basically three kinds of files supported by the system: ordinary disk files, directories, and special files. Ordinary files contain whatever information the User places on it. Source programs, object modules generated by compilers or assemblers, and pure text are examples of ordinary files. Directories provide a mapping of ordinary file names to the files themselves, thereby inducing a structure on the file system. All files in the system may be located by tracing a path from the "root" directory to the desired file. User interface to each input/ouput (I/O) device supported by the system is through a device-specific special file. With this scheme, I/O devices can be treated as ordinary files.

B. MEMORY MANAGER DESIGN

The UNIX memory manager is, in concept, a relocatable partitioned memory manager with swapping and limited segmentation. While the processor is executing in behalf of a process, the process image must reside in a contiguous region in main memory. As already explained, during the execution of other processes, a process may remain in memory unless scheduling of a higher priority process forces it to be swapped out (copied) to the swap device. UNIX processes are logically divided into three parts: the UVECTOR, the

User data space, and the User processor stack. If a process requires use of shared text, its data space contains only data. Shared text is managed separately.

The UVECTOR contains all process status information required by UNIX while the process is resident in core. Other process status information, which must remain addressable throughout the life of the process, is contained in a system control block called a PROC. In the case of shared text, the PROC contains a pointer to yet another system control block called a TEXT. This block contains all information necessary to control the sharing of a text segment by one or more processes. Shared text, if required, is established in memory independently of the processes which are sharing it. If a process shares text, "exec" checks to see if a copy of the text segment is already available in the system. If it is not, a copy is created.

User address space of a text-sharing process may be created with separate instruction space (I-space) and data space (D-space), or with combined I-space and D-space. User address space for non-sharing processes is established with combined instruction and data spaces. If the I-space and D-space of a process are combined, there is no differentiation between instructions and data. The file type [7] of an executable file is used by "exec" to establish a separation flag in the UVECTOR. This flag is used to control the method by which the address space for a text sharing process is established. The shared text segment of a process with separate address spaces is

addressed beginning at User I-space address 0; data is addressed beginning at User D-space address 0. If a process with combined address spaces has shared text, the text segment is addressable beginning at address 0 in both I-space and D-space. Its data is addressed in both I-space and D-space, beginning at the first 4K word boundary above the shared text segment. For processes without shared text, the text is addressed beginning at address 0 in the combined I-space and D-space; and its data is addressed beginning at the first word boundary above the text. The User's processor stack is addressed extending downward from the highest address in D-space or combined I-space and D-space.

PDP-11/50 page address registers (PARs) and page descriptor registers (PDRs) are loaded when a process image is brought into memory or its address space is reestablished. PARs are used by the MMU to translate virtual addresses to physical memory addresses. PDRs are used to describe a set of attributes about a resident process's pages. A page in UNIX can be thought of as a partition of a process image which can be up to 4K words in length. Access control specified in the PDRs is read only for shared text pages and read-write for all other pages.

User data space may vary dynamically if required during execution of a process. A system call, "break", is provided in UNIX to alter the size of the data area. Additionally, the size of a process image may be increased by dynamic growth of the User processor stack. When this occurs, the system automatically increases the amount of space provided

for the process. In both of these cases, UNIX must reestablish the entire process image in main memory. As mentioned previously, one of the primary benefits of segmentation is indicated here. SUNIX and SSUNIX require that only the segment which changes size be reestablished.

1. Segmentation

Segmentation of shared text is already well supported in UNIX. SUNIX and SSUNIX further divide process images into the three natural segments mentioned before: UVECTOR, data space (including non-shared text), and User stack. Allocation of memory space for a process is accomplished by separately establishing each segment in a free area large enough for it. Contiguous allocation in memory is possible, but not required. Swap space for a process is still allocated in a contiguous partition on the system's swap device. However, due to the separation of segments in main memory, up to three copy operations are required each time a process is swapped to or from the swap device. In UNIX, only one copy operation is required. Thus, there is an increase in system overhead with segmented memory management. There is, however, a compensating advantage to segmentation: a reduction of overhead (segment copying) results from independent management of data and stack segments if dynamic growth of one of these segments is required.

2. Allocation of Memory Space

The basic quantum of memory space in UNIX is a 64-byte area called a block. This is the smallest quantum supported by the PDP-11/50 MMU. Memory space for each process is assigned from a free memory map maintained by the operating system. This map, which is established at system initiation time based on physical memory configuration, contains the base address and size (in blocks) of each unallocated area of User memory space. Addresses in the map are increasing in order and represent the physical block number of the free area to which they refer. The operating system is established beginning at physical address 0. User memory space begins at the first block boundary above the system code.

Maintenance of the free memory map is performed by the memory management primitives "malloc" and "mfree". These system routines are also generalized to perform maintenance of all other system free maps; for example, the swap map, and the shared memory maps (described later) in SSUNIX. The function of "malloc" is to locate an area of given size in the given map, update the map to reflect the allocation of the area, and then return the physical block number of the allocated area to the calling routine. The algorithm used in "malloc" is "first-fit". That is, the free map is searched sequentially, beginning with the first entry, until an area of sufficient size to satisfy the request is found. If the requested space is not available, a zero value is returned to the caller. The function of

"mfree" is to free the area specified by the calling routine. The region specified is sorted back into the free map and combined on one or both ends if possible. In UNIX, processes are allocated space based on the size of the entire process image. In SUNIX and SSUNIX, each segment of a process is allocated memory space separately based on the segment size.

3. Multiported Memory

Regions of multiported memory are often desirable to support peripheral devices which have unique DMA requirements. Examples are devices which have hardware address range limitations or require real-time memory access. Specific applications of multiported memory in the Signal Processing and Display Laboratory have already been discussed.

To support multiported memory, the operating system must be able to align a process or process segment in the multiported memory region. UNIX does not have this capability. As explained in the previous section, "malloc" assigns memory space from the User free map based on a "first-fit" algorithm. With this algorithm, alignment within a specified region of memory would be purely coincidental. Furthermore, since DMA by a peripheral device to UVECTOR or stack addresses violates system security, only the data segment is placed in the multiported region. This implies that segmentation is a desirable memory management scheme in a system configured with multiported memory. For

this reason, SUNIX was chosen as the basis for development of multiported memory system support. Although there was no provision for data segment alignment in SUNIX, the method of process segmentation directly supported the design effort undertaken by this author.

In order to investigate multiported memory applications in UNIX, a shared-segmented memory manager (SSUNIX) was designed and implemented. SSUNIX provided the capability to align a calling process's data segment in a shared memory region, and to ensure system protection by clearing all other processes from that region. Free memory maps were established and maintained for each shared core area. Processes requested the shared memory asset via a system call ("getshr"). A system routine was designed which checked the User free memory map to determine if the requested region was free. If the area was free, it was removed from the User free memory map and the data space of the calling User process was moved to the shared region. The "malloc" system call was used to allocate space for the data segment in the requested shared region. If the area was in use by another memory-sharing process, the data segment of the calling process was simply allocated in the shared memory map and then moved to the region. If the area was in use by other non-memory-sharing processes, these processes were swapped out of memory until the area was clear. The data segment of the calling User process was then allocated in the shared memory map and moved to the region.

In each of the above cases, after a shared memory region had been acquired for a memory-sharing process, non-memory-sharing processes were not allocated space within the bounds of the shared region. Additionally, the calling process was locked into memory so it would not be swapped out of the area. Another system call ("freeshr") was implemented to release a process's shared memory asset and unlock the process image. This system call updated the shared memory map. If no other memory-sharing processes remained in the region, the entire shared memory area was returned to the User free map.

IV. MODIFICATIONS TO UNIX

A. GENERAL

Modifications to the UNIX operating system to produce SUNIX were performed by Emery [4] in mid-1976. The approach taken in the design was to make the general structure of memory management familiar to those who understood the structure of UNIX. This approach led to a well-designed operating system which readily supported further modifications and debugging. However, due to greater emphasis on the completion and testing of a partitioned segmented memory manager (PSUNIX), SUNIX was not implemented at that time. Final implementation and testing of SUNIX was a primary goal of this thesis project. SSUNIX was to be a direct extension of SUNIX. For this reason, modifications to UNIX required to implement SSUNIX encompass those made for SUNIX.

The changes necessary to implement process segmentation with data segment alignment affected five general areas:

1. Control Blocks
2. Memory Management
3. Swap Space Allocation
4. Shared Memory Allocation
5. Support Programs

Each of these areas is discussed in a subsequent section of this chapter. The appendices to this thesis provide further documentation. Information on control block modifications is found in Appendix A. Memory management and shared core allocation modifications are described in Appendix B.

B. CONTROL BLOCKS

UNIX control blocks are data structures used by the operating system to maintain information vital to system control. The only control block modification required to support process segmentation was the PROC. A list containing a PROC for each active process is maintained by the system. In UNIX, a PROC contains the block address of the UVECTOR and the size of the process image (which does not include shared text). In SUNIX and SSUNIX, a PROC contains the block address of each of the process segments and the sizes of the data and stack segments.

Several new control blocks were added to implement data segment alignment within a shared core area. SHMEM defines the upper and lower bounds of each shared memory region. System shared core configuration changes require modifying the entries in this structure. SHMEM is used at system initiation time to initialize two other shared memory control blocks. A SHARE contains information about a particular shared-memory asset. The high and low bounds of the region, a flag indicating whether or not the region is in use by another memory-sharing process, and another flag

used by "malloc" are maintained here. A SHRMAP is a structure which contains a free memory map for each of the shared regions. The maps in SHRMAP were configured exactly like the other system free maps so they could be maintained by "malloc" and "mfree".

C. MEMORY MANAGEMENT MODIFICATIONS

Adapting the existing UNIX memory management routines to perform process segmentation was a straightforward problem: in each routine that dealt with a process image, the process image was managed as three independent segments instead of a single entity. Although simply stated, solving this problem required many hours of familiarization with existing UNIX concepts, and modifying several hundred lines of source code. Details of these modifications are not included here, but are contained in Appendix B. Copies of system source code cannot be included in this thesis due to UNIX non-disclosure agreements. However, authorized UNIX User's may acquire machine-readable copies by contacting the Naval Postgraduate School.

D. SWAP SPACE ALLOCATION MODIFICATIONS

Swap space is allocated to a process when it must be swapped out of memory, and freed when the process returns to memory. A map of free space on the system's swap device is maintained by "malloc" and "mfree". The system routine "swap" is used to transfer data between memory and the swap

device. SUNIX and SSUNIX processes consist of three segments that are independently established in main memory. Moving a process image to the swap device requires a "swap" operation for each segment. Also, when shared text must be established, an additional "swap" operation is involved. Actual space on the swap device is allocated in a contiguous block. The disk address of the process is the address of the UVECTOR. Data (if any) and stack are located immediately following the UVECTOR. Shared text is separately established, and retains it swap space until there are no longer any live processes which require it.

E. SHARED MEMORY ALLOCATION MODIFICATIONS

The basic ideas underlying the design of a shared-segmented memory manager for UNIX (SSUNIX) were presented in Chapter III. Actual design work was started after SUNIX was implemented and had demonstrated satisfactory performance. The approach taken in the development of SSUNIX was to add several system level routines to perform the shared memory management functions, and thus modify existing SUNIX code as little as possible. Three new system routines were added: "ckmap", to check the User free memory map; "shalloc", to allocate a process's data segment to a given shared core region; and "shfree", to free a process's shared core asset. System calls for User program interface with "shalloc" and "shfree" were also implemented. From a User's program, "getshr" acquires a shared core asset, and "freeshr"

releases it. A brief summary of each new routine and the other modifications made to SUNIX is provided in the following paragraphs. Detailed information can be found in Appendix B.

The new procedure "ckmap" was the basic primitive used for shared core allocation. Its function was twofold: check the User free memory map to determine if an area between a given high and low block address was free, and then, if the area was free, remove it from the free map. The design of this procedure provided an interesting challenge. Once it was determined that the area was free, several boundary alignment conditions had to be considered in order to remove it from the free memory map. The procedure "shalloc" performed the actual allocation and assignment of a processes data segment to the requested shared core area. It also locked the process image in memory. The algorithm implemented was described in the previous chapter. In order to prevent allocation of another process within the shared core region during the shared core acquisition by "shalloc", a slight modification to the memory management primitive "malloc" was required. A process's shared memory asset was freed by "shfree". In addition to the system call interface, this routine was called from the "exit" procedure to ensure that a process's shared core was released at process termination.

Other modifications to SUNIX included: "main", to establish shared core regions and initialize their respective control blocks; and "sysent", to provide system

entry points for the system calls "getshr" and "freeshr". The source code required to build both SUNIX and SSUNIX can be found in the directory /usr/sys.sunix on the display processor's file system.

F. SUPPORT PROGRAM MODIFICATIONS

One support program modification was required for SUNIX and SSUNIX. The Shell command "os" [11] displays certain information about active process status. It uses the PROC list to acquire this information for display at the User's terminal. Since the structure of the PROC was modified to support process segmentation, certain variables used by "os" were invalidated. In particular, the address and size of a process image in UNIX were described in two variables in the PROC. In SUNIX and SSUNIX, five variables were required: UVECTOR address (its size is fixed at .5K words), data segment address, data segment size, stack segment address, and stack segment size. To retain the display format used by "os", it was decided that all five of these attributes would not be presented. Since the User is normally concerned with only his data segment's address and size, these values were substituted for the existing values in "os". The revised version of this routine can be found in /usr/sys.sunix on the display processor's file system. The object version of "os" found in /bin must be replaced with the revised object during SUNIX and SSUNIX operation.

V. EVALUATION OF PERFORMANCE

A. OVERVIEW

As stated throughout this report, completion and testing of SUNIX was an integral part of this thesis and a prerequisite for the development of SSUNIX. A severe problem with the SUNIX memory manager had prevented its final implementation. This problem manifested itself in an inability to assemble certain programs. Since the assembler is used during a pass of the "C" compiler, compilations were also affected. Several weeks at the outset of this thesis project were required to become familiar with, debug and test SUNIX. Once familiar with the concepts of system organization and memory management, Emery's design readily supported the debugging effort at hand. The problem was eventually found and corrected in the memory management routine "exec". Development, implementation and testing of SSUNIX followed. Performance of SUNIX and SSUNIX was evaluated in relation to UNIX. Additionally, tests were conducted which demonstrated the capability of SSUNIX to perform data segment allocation and deallocation of shared core.

B. COMPARISON WITH UNIX

The method chosen to compare performance of UNIX, SUNIX and SSUNIX was to use the elapsed execution time of a standard stream of processes (benchmarks). A series of benchmarks was run to determine relative performance under a variety of operating conditions. Available User memory was the variable used to establish these conditions. Two benchmarks were used: a monoprogramming benchmark, BENCH1; and a multiprogramming benchmark, BENCH2. These benchmarks are essentially identical to those used by Emery in his testing of PSUNIX. Both BENCH1 and BENCH2 contain the same sequence of UNIX commands. These commands are documented in Ref. 11. APPENDIX C contains benchmark listings. The computer system used for the tests was the multiuser side of the laboratory configuration shown in Figure 1. A single-user environment was established with only the console on-line. The purpose of this was to prevent the introduction of an added variable in benchmark performance due to processes generated by other Users on the system. The swap space and file system used were located on RP05 disk units [3].

Table I presents the results of benchmark tests for UNIX, SUNIX and SSUNIX. Available User memory space was varied to evaluate performance over a wide range of system configurations. Timing data was obtained by using the UNIX "time" command [11]. Processes run under control of "time" are clocked by sampling processor state at the rate of 60

Hz. "Real" time reflects the total elapsed time for the process and is reported to the nearest second. "User" time is the time spent in the User state (ie. executing the User's program instructions) and is reported to the nearest tenth of a second. "Sys" time is the time spent during the execution of operating system supervisory instructions and is reported to the nearest tenth of a second. The difference between "real" time and the sum of "user" and "sys" times indicates the amount of processor idle time. Idle time generally reflects the amount of time spent on asynchronous I/O operations.

In addition to benchmark testing in a single-user configuration, SUNIX was run for a full day of multiuser program development. The system was fully configured (Fig. 1) and moderate to heavy system utilization was noted. System Users were asked to report any problems to the laboratory staff. No system failures occurred and no problems were reported. This test provided an excellent indication of proper system operation as well as the transparency of the memory management method to system Users.

C. SHARED MEMORY ALLOCATION

Testing of data segment allocation to shared core was performed on the display processor. The system was configured with a single terminal and the system console. Swap space and the file system were maintained on RP05 disk

units. To demonstrate the performance of SSUNIX, it was necessary to develop a method for displaying certain relavent system information. Three items were required: the location of a process's data segment in memory, the state of the User free memory map, and the state of each of the shared memory maps. This data best indicated the shared memory allocation/deallocation process. A system call ("shtest") was developed to display this information at the display processor's console. Several test programs which included shared core allocation requests ("getshr") and shared core deallocation requests ("freeshr") in various sequences were developed. "Shtest" was included in these programs to dislay the required information after each shared core operation. This approach proved invaluable in the debugging and refinement of SSUNIX.

D. ANALYSIS OF RESULTS

The results shown in Table I clearly indicate that the performance of UNIX, SUNIX and SSUNIX are nearly identical. No statistical analysis was performed on these results, but earlier work [6] indicates that the sampled data may have a standard deviation of as much as 8 percent on identical benchmarks run several times on the same system. System supervisory times ("SYS") under SUNIX and SSUNIX were found to be slightly better than UNIX in some cases. This is a surprising result in light of the more complex memory management function with process segmentation. The

disadvantage related to increased swapping overhead did not appear to degrade overall system performance. A probable explanation for this result is an offsetting performance improvement due to independent dynamic growth of data and stack segments. These factors were discussed in Chapter III. SSUNIX's performance in a multiported memory environment was validated. The tests performed using "shtest" indicated that shared core allocation and deallocation of a process's data was properly managed. Additionally, since SSUNIX benchmark timing data was so nearly identical to that of the other systems, the data segment alignment capability had no significant effect on system performance.

	UNIX	SUNIX	SSUNIX
REAL	2:25.0	2:26.0	2:25.0
USER	1:24.7	1:24.2	1:24.4
SYS	31.3	31.0	31.8

BENCH1, 48K Words

	UNIX	SUNIX	SSUNIX
REAL	2:10.0	2:10.0	2:13.0
USER	1:25.9	1:25.6	1:26.9
SYS	36.4	35.3	34.6

BENCH2, 48K Words

	UNIX	SUNIX	SSUNIX
REAL	2:11.0	2:11.0	2:12.0
USER	1:25.7	1:27.1	1:26.6
SYS	34.8	34.7	35.3

BENCH2, 40K Words

	UNIX	SUNIX	SSUNIX
REAL	2:16.0	2:21.0	2:23.0
USER	1:27.0	1:26.8	1:27.3
SYS	34.3	34.2	35.3

BENCH2, 32K Words

Table I. Benchmark Timing Data

VI. CONCLUSIONS AND RECOMMENDATIONS

The primary goal of this thesis, an extension of the UNIX memory manager to provide data segment alignment with system protection, was successfully implemented in the form of SSUNIX. Performance results were encouraging. However, before the benefits of this system can be fully realized, other related development work in the Signal Processing and Display Laboratory is necessary. This chapter provides recommendations directed toward the development of a total system support package for the laboratory's display processor. Also included is a recommendation for further research in the area of segmented memory management.

A. CONTROLLED MEMORY ALLOCATION

Several different operating systems for the display processor have been developed to provide support for special peripheral devices. Two of these systems were designed to provide process alignment in a particular area of memory: one for CSP 125 processes, and another for VG processes. The existence of a number of different operating systems, each with a particular application, causes a significant configuration control problem in the laboratory. Since SSUNIX was designed with these specific applications in mind, a direct implementation of this system on the display

processor would simplify the system configuration control problem.

B. MULTIPROCESSOR INTERFACE

As indicated in Figure 1, the PDP-11/34 slave processor is presently treated as an independent system peripheral. The design work done by Gray [5] has not yet been tested in the multiprocessor environment for which it was intended. Additionally, the hardware interfaces for the 32K word shared core area and the VG display device have not been implemented. The dashed lines in Figure 1 depict the configuration required to complete the multiprocessor interface. Once this is done, implementation of the slave processor's operating system and integration of SSUNIX into the multiprocessor environment could follow. Investigation of the communication requirements and protocol between the master processor and the slave processor is required. Also, the inter-processor graphics support package designed by Visco [12] must be reviewed to ensure proper interface with the "getshr" and "freeshr" system calls of SSUNIX. Only slight modification to SSUNIX would be required to establish this interface. One suggested approach is to implement Visco's "rtime" and "nonrtime" system calls in SSUNIX to perform the functions as "getshr" and "freeshr". This approach has the advantage of requiring only simple modifications to SSUNIX and no modifications to the graphics interface routines. Another possibility is modifying the

Vector General routines to acquire shared core via "getshr", and release it via "freeshr".

C. INVESTIGATION OF SWAPPING POLICIES

The disadvantage of segmented memory management requiring a greater number of copy operations for process swapping has been stated previously. An interesting topic for further investigation is the development of an extension to SSUNIX (or SUNIX) which implements swapping on a segment basis. Modifications to SSUNIX to support this investigation would involve revision of the process scheduling routine "sched", the process memory allocation routine "pralloc", the routine for swapping processes out of core, "xswap", and the routine for swapping processes into core, "prswap". It is contended that, although the memory management function will be slightly more complex, the actual changes required to existing routines will not degrade system performance. In fact, since swapping is a significant factor in system overhead, an optimum swapping policy should enhance overall performance.

APPENDIX A: CONTROL BLOCK MODIFICATIONS

A. DOCUMENTATION FORMAT

This appendix is intended to be used with a copy of the source code for UNIX (Version 6), SUNIX and SSUNIX. It contains documentation of the modifications made to UNIX control blocks to implement the two new segmented systems. Source code for these control blocks in the UNIX form is found in the directory /usr/sys. Source code for the modified versions can be found in the directory /usr/sys.sunix on the display processor's file system. The format of this appendix and some of the documentation contained herein are identical to APPENDIX A of Ref. 4. Each control block is described under an upper case roman letter. The control block name is followed by the source code file in which it is found. An overview and a description of significant data elements is provided for each control block.

B. SHRMAP, share.h

1. Overview

SHRMAP is a structure containing NSHARE free memory maps of shared core regions. NSHARE is a tunable parameter, also found in "share.h", which specifies the number of shared core areas in the system. This control block is not

found in UNIX or SUNIX. Each map in SHRMAP is an integer array containing the physical block number and size of an unallocated memory area. The maps are sorted in physical block number order. This configuration is identical to COREMAP and SWAPMAP. Documentation of "malloc.c" in APPENDIX B describes the memory management primitives which manipulate the free memory maps.

2. Significant Data Elements

a. int shmap[SHMAPSIZ]

This is the specification for a single shared memory map contained in SHRMAP. SHMAPSIZ is a tunable parameter defined in "share.h".

b. char *m_size

This is the size of the free area in 64-byte blocks.

c. char *m_addr

This is the physical block number of the beginning of the free area in the shared core region. If this value is zero, it marks the end of the map.

C. SHARE, share.h

1. Overview

A SHARE contains certain control information about a shared core region. There is one of these control blocks for each of the NSHARE regions. Share is not defined in

UNIX or SUNIX.

2. Significant Data Elements

a. int basaddr

This is the physical block number of the base of the shared core region in memory.

b. int hiaddr

This is the physical block number of the top of the shared core region.

c. char nuseflg

This is a flag which indicates that the shared core region is in use by a resident memory-sharing process. Further details on its use are provided in appendix B under the documentation of "shalloc.c".

d. char ckflg

This is a flag which indicates that no processes are to be allocated memory space within the bounds of the shared region. Its use in "malloc()" is documented in APPENDIX B.

D. PROC, proc.h

1. Overview

One PROC is allocated for each active process in the system. The PROC exists for the life of the process. PROCS are maintained in an array called "proc" which is NPROC in size. This array is permanently resident in main memory and

contains all per process information which cannot be swapped out of main memory.

2. Significant Data Elements

a. char p_flag

This is a word of flags. Bit 0 of this word is the SLOAD flag. If it is set, the process is resident in main memory. Bit 1 is the SLOCK flag. If it is set, the process is locked in memory and may not be swapped out. In SSUNIX, this bit is set to lock memory-sharing processes in memory. Bit 2 of this word is the SSWAP flag. If it is set, the process is being swapped out.

b. int p_addr

This variable is present only in UNIX. If the process is resident in main memory, it is the physical block number of the process's UVECTOR. If the process is swapped out, it is the swap device block number of the swapped image.

c. int p_size

This variable is present only in UNIX. It is the size of the process's swappable image in 64-byte blocks.

d. int p_textp

This is a pointer to the process's TEXT. If the value is zero, the process does not have shared text.

e. int p_caddr

This variable is present only in SUNIX and SSUNIX. It is the main memory physical block number of the process's UVECTOR while the process is in memory.

f. int p_daddr

This variable is present only in SUNIX and SSUNIX. It is the swap device block number of the process's swap space. If it is zero, the process has no swap space.

g. int p_dsize

This variable is present only in SUNIX and SSUNIX. It is the size of the process's data segment in 64-byte blocks.

h. int p_ssize

This variable is present only in SUNIX and SSUNIX. It is the size of the process's stack in 64-byte blocks.

E. UVECTOR, user.h

1. Overview

The structure "user" is referred to as the UVECTOR. One of these structures is part of each swappable process image. The UVECTOR contains all process data that is not needed when the process is not in control of the processor. When the process is in control of the processor, its UVECTOR resides at a fixed location in the operating system data space.

2. Significant Data Elements

a. int u_uisa[16]

In UNIX this array contains the 64-byte block displacements from the start of the region of the process's data and stack pages. In SUNIX and SSUNIX this array is not used.

b. int u_uisd[16]

This aray contains the prototypes of the process's user I-space and D-space page descriptor registers.

c. int u_tsize

This is the size of the process's shared text segment in 64-byte blocks.

d. int u_dsize

This is the size of the process's data segment in 64-byte blocks.

e. int u_ssize

This is the size of the process's stack segment in 64-byte blocks.

I. APPENDIX B: MEMORY MANAGEMENT MODIFICATIONS

A. DOCUMENTATION FORMAT

As with APPENDIX A, this appendix is intended to be used with a copy of the source code for UNIX, SUNIX and SSUNIX. It contains documentation of the modifications to UNIX memory management functions which were required to produce the two segmented systems. The format used here, and some of the documentation, is identical to APPENDIX B of Ref. 4. UNIX source code is divided into several files containing related blocks of code. The functions documented in this appendix are grouped under an upper case roman letter and the name of the file in which they are found. Each function in each file is labeled with an arabic number. The documentation of each function is divided into four subsections: parameters, functional description, returned values, and error conditions. The files containing the UNIX functions are found in the directory /usr/sys/ken. The files containing the modified versions of the functions for SUNIX and SUNIX are found in the directory /usr/sys.sunix/ken on the display processor file system. In this directory, those files which contain modifications specific to SSUNIX are prefixed with the letters "sh".

B. main.c

1. main()

a. Parameters

This function has no parameters.

b. Functional Description

This is the operating system initiation function. Physical memory configuration is determined, User memory space is cleared, and all system free maps are initialized. Process 0 and Process 1 are created. In SSUNIX, the specification array for shared core configuraton, "shmem", is found in the file "main.c". This function utilizes "shmem" to initialize the SHRMAPs and SHAREs described in APPENDIX A. Shared core configuration changes are managed by modifying the entries in "shmem". Upon completion of initiation tasks, the process scheduling routine, "sched", is called. "Sched" then runs until the operating system crashes or is otherwise terminated.

c. Returned Values

This function does not return.

d. Error Conditions

An error occurs if the system clock cannot be established.

2. estabur(nt,nd,ns,seo)

a. Parameters

The first three parameters are the sizes of the current process's shared text, data and stack segments in 64-byte blocks. The last parameter is a separation flag that is set if the process has split instruction and data spaces. The current process's UVECTOR is an implied parameter.

b. Functional Description

This function first checks the validity of its arguments. It loads the prototypes of the memory management page descriptor registers into the array u+uisd[] found in the current UVECTOR. In UNIX it also loads page start displacements measured in blocks from the beginning of the region or text into the array u+uisa[] found in the current UVECTOR. The array u+uisa[] is not loaded in SUNIX and SSUNIX. Its values are are not required since the values of the parameters are placed in the variables u+tsize, u+dsize, u+ssize, and u+seo in the current UVECTOR. In all versions, "sureg" is called to load the actual memory management registers.

c. Returned Values

If the parameters are invalid, minus one is returned; otherwise, a zero is returned.

d. Error Conditions

The minus one return indicates an error to the caller.

3. cksize(nt,nd,ns,sep)

a. Parameters

The parameters are the same as for "estabur(nt,nd,ns,sep)" described above.

b. Functional Description

This function is present only in SUNIX and SSUNIX. It checks its parameters to see if they are valid.

c. Returned Values

If the parameters are invalid, a minus one is returned. Otherwise, a zero is returned.

d. Error Conditions

A minus one return indicates an error to the caller.

C. malloc.c

1. malloc(mp,size)

a. Parameters

The parameters are a pointer to a free memory map array and the size in blocks of the region to be allocated from the mac.

b. Functional Description

This function allocates space in main memory and on the swap device. If User free memory space is to be allocated, the first parameter must point to COREMAP, and the size must be specified in 64-byte blocks. If swap space is to be allocated, the first parameter must point to SWAPMAP, and the size is specified in 256-word sectors. If shared core is to be allocated, the first parameter must point to a SHRMAP, and the size is specified in 64-byte blocks. In SSUNIX only, this function checks the global flag, "sharflg", if COREMAP is specified. If "sharflg" is set, the function must then check the "ckflg" in each SHARE before allocating space in COREMAP. If a "ckflg" is set, a free area which includes any part of that particular shared region will not be allocated.

c. Returned Values

If allocation is successful, this function returns the physical block number of the base of the allocated area. Zero is returned if space is unavailable.

d. Error Conditions

A zero returned value indicates allocation failure to the caller.

2. mfree(mp,size,aa)

a. Parameters

The first two parameters are the same as those for "malloc" described above. The third parameter is a physical block number of an area of main memory or swap space.

b. Functional Description

This function frees the specified area in the specified free map. If the first parameter points to COREMAP, the area is freed in the User free memory map. If the first parameter points to SWAPMAP, the space is freed in the free map of the swap device. In SSUNIX only, if the first parameter points to a SHRMAP, the area is freed in the map of that particular shared core region. Sizes are in 64-byte blocks if COREMAP or a SHRMAP is specified, and in 256-word sectors if SWAPMAP is specified.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

This function has no error conditions.

3. ckmap(rla,rha)

a. Parameters

The parameters are the low address and high address in physical block numbers of a region of main memory space. COREMAP is an implied parameter.

b. Functional Description

This function is present in SSUNIX only. It checks COREMAP to determine if the specified region of main memory is free. If the area is free, it is removed from COREMAP and the map is rebuilt to reflect the removal. In order to rebuild the map, several boundary alignment conditions are checked. The position of the free area boundaries in relation to the boundaries specified in the parameters determines the algorithm used to sort and rebuild the map. This function is used in conjunction with the shared core allocation function, "shalloc", described below.

c. Returned Values

If the specified area is free and was removed from COREMAP, a one is returned. If any part of the area is in use, a zero is returned.

d. Error Conditions

This function has no error conditions.

4. shalloc(scname)

a. Parameters

The parameter is an integer value which specifies a particular shared core region. The PROCs and TEXTs of all processes, the current process's UVECTOR, the SHAREs, and the SHRMAPs are all implied parameters.

b. Functional Description

This function is present only in SSUNIX. It performs the shared core allocation process by acquiring the region, reserving it for memory-sharing processes, and positioning the caller's data segment in it. The function first checks the validity of its input argument. The "ckflg" for the specified region and the global "sharflg" are set to prevent allocation of space within the region during the acquisition process. To ensure that the calling process is not within the bounds of the shared core area that it is trying to acquire, it is swapped out of memory using "ceswap". Since "malloc" will not allocate space within the region, when the process returns to memory it will not interfere with its own acquisition process. The "nuseflg" of the shared core area is checked to determine if the area has been previously acquired and reserved for sharing. If the area has been reserved, the caller's data segment is allocated in the SHRMAP and then copied to the allocated space. If the area is not reserved, "ckmap" is called to determine if it is free. If it is not free, the PROCs are scanned to find processes with segments in the area. Interfering processes are swapped out of memory until the region is free. "Ckmap" performs the actual reservation process by removing the region from COREMAP. The caller's data segment is then allocated in the SHRMAP and copied to the allocated space. In any case, after the process's data segment has been copied to the region, the process image is locked in main memory by setting the SLOCK flag in the PROC.

c. Returned Values

If the shared core allocation process is successful, the physical block number of the base of the caller's data segment is returned. If unsuccessful, a minus one is returned.

d. Error Conditions

The minus one return indicates an error to the caller. This value can result from one of two error conditions: improper input parameter, or failure to allocate the process's data segment to the shared core region.

5. shfree(p)

a. Parameters

The parameter is a pointer to the calling process's PROC. The SHAREs and SHRMAPs are implied parameters.

b. Functional Description

This function is present only in SSUNIX. It is used to free the caller's shared core asset. It first checks to see if the caller is a memory-sharing process. If so, that process's data segment is deallocated in the SHRMAP of the region that it occupies. If there are no other memory-sharing processes remaining in the area, it is freed in COREMAP and the caller's image is unlocked. If any memory-sharing processes remain, the caller's image is unlocked and swapped out of memory. In this case, the space occupied by the data segment is not freed in COREMAP since

the area must remain reserved.

c. Returned Values

The values returned by this function are not checked.

d. Error Conditions

This function has no error conditions.

D. sig.c

1. core()

a. Parameters

The current process's UVECTOR, PROC, TEXT and address space are implied parameters.

b. Functional Description

This function creates a memory image of the current process's UVECTOR, data, and stack. In UNIX this function uses "estabur" to redefine the process's virtual address space and make the data and stack contiguous. It then writes the data and stack in one output operation. In SUNIX and SSUNIX this is impossible because the data and stack may not be physically contiguous. Two output operations are required; one for the data and one for the stack. If an error occurs during an output operation, an indication is left in u.u_error in the UVECTOR of the current process.

c. Returned Values

This function returns zero if successful and one if an output error occurs.

d. Error Conditions

The one return indicates an error to the caller.

2. grow(sp)

a. Parameters

The parameter is the value of the current process's User stack pointer. The current process's UVECTOR and PROC are implied parameters.

b. Functional Description

This function is called asynchronously when the current process's stack attempts to expand byond the space allocated for it. This function calculates the number of blocks that the stack must be increased, validates the new stack size, and acquires the memory that is needed. In UNIX, "expand" is called to establish the new, larger address space for the entire process image. In SUNIX and SSUNIX, this function attempts to acquire the needed space by in-core expansion of the stack segment. If unsuccessful, it calls "ceswap" to acquire the space by swapping. If successful, it copies the old stack to the new space and frees the old memory. In all versions the newly acquired space is cleared and "estabur" is called to reload the memory management registers.

c. Returned Values

This function returns a zero if it is unsuccessful and a one if it is successful.

d. Error Conditions

A zero return indicates an error to the caller.

E. slp.c

1. sched()

a. Parameters

The PROCs and TEXTs of all processes are implied parameters.

b. Functional Description

This function searches for swapped out processes that "deserve" to be returned to memory. It selects the most "deserving" process; acquires space for it by swapping out other processes if necessary; and returns it to main memory. In SUNIX and SSUNIX two new functions are used: "mralloc" to acquire main memory for the process, and "mrswap" to swap it in.

c. Returned Values

This function does not return. It is the basic instruction loop for Process 0. It goes to sleep and is reawakened about once per second by the clock function.

d. Error Conditions

In UNIX, if a swap input or output error occurs, the message "swap error" will be sent to the console and the system will crash. In SUNIX and SSUNIX, the swap operations occur in "prswap" so no error messages are generated here.

2. newproc(nrp)

a. Parameters

The parameter is a pointer to a PROC to be established for a child process. The current process's UVECTOR, PROC, and TEXT are implied parameters.

b. Functional Description

This function creates an exact duplicate of the current process as a child of the current process. It first makes the appropriate entries in the child and parent PROCs and in the TEXT if one exists. It then attempts to acquire memory for the child process. If it is successful, it simply copies the parent's image to the new memory. If it fails, it swaps out a copy of the parent to be returned to memory as the child. In SUNIX and SSUNIX, a new function, "pralloc", is used to attempt to acquire memory for the child.

c. Returned Values

This function returns zero to the parent process. The return to the child does not come from this function, but from the scheduling function "swtch". The child can identify itself as the child because "swtch"

returns a one to it. This is one of the most important and subtle phenomena in UNIX.

d. Error Conditions

If the PROC pointed to by the parameter is already allocated to an active process, the message "no procs" will be sent to the console and the system will crash.

3. expand(newsize), expand(newd,news)

a. Parameters

In UNIX, this function is called with a single argument, the new region size. In SUNIX and SSUNIX, this function is called with a pair of arguments: the new data size and the new stack size. The current process's PROC and UVECTOR are implied parameters.

b. Functional Description

In UNIX, this function is called whenever the size of the current process's memory image changes. It puts the new size in p_size in the current PROC. If the new size is smaller, it simply frees the unwanted memory. If the new size is larger, it attempts to acquire the new space for the process. If it succeeds, it copies the process image to the new area. If it fails, it causes the process to be swapped out with the new sizes noted in the PROC. When it returns to memory, it will return at the new size. If the process is swapped out, this function calls "swtch" to reschedule the process immediately. In SUNIX and SSUNIX, it puts the

two new sizes in p_dsize and p_ssize in the PROC. In UNIX, the function calls "xswap" to perform the swapping operation. In SUNIX and SSUNIX, the new function "ceswap" is used. In UNIX, "sureg" is called to load the memory management registers. In SUNIX and SSUNIX this is not necessary.

c. Returned Values

The return values of this function are not checked. The caller has no way of knowing if the process was increased in size by direct expansion or by swapping. In UNIX, if the process is swapped out, this function does not return to its caller. The return comes from a subsequent call to "swtch" after the process has returned to memory.

d. Error Conditions

This function has no error conditions.

4. ceswap(ods,oss)

a. Parameters

The parameters are the current process's data and stack segment sizes in 64-byte blocks.

b. Functional Description

This function is present only in SUNIX and SSUNIX. It is called to perform core expansion swapping. It calls "xswap" to do the actual swapping and then calls "swtch" to reschedule the process immediately.

c. Returned Values

This function does not return to the caller. The return comes from a subsequent call to "swtch" after the process has returned to memory.

d. Error Conditions

This function has no error conditions.

F. sys1.c

1. exec()

a. Parameters

The current process's UVECTOR, PROC, and TEXT are implied parameters. Because this function is a system call, the array u_uarg[] in the UVECTOR contains additional arguments. See Ref. 11.

b. Functional Description

This system call is used by the current process to invoke a new program. It copies any program arguments to a buffer, unlinks from the old TEXT, frees its old main memory space, establishes a new TEXT if the new program has shared text, acquires memory space for the new data and stack segments, clears the region acquired, reads in the new program's data, copies the arguments to the new stack, and changes the memory management registers to reflect the new address space. In UNIX, "expand" is called to free the old memory space; in SUNIX and SSUNIX, a new function, "prfree", is used. In UNIX, "estabur" is used to validate the new

memory requirements; in SUNIX and SSUNIX, the new function "cksize" is used. In SUNIX and SSUNIX, only the uninitialized portion of the data segment is cleared before the copy operation.

c. Returned Values

This system call returns to the caller only if it encounters an error. If no error occurs, it returns to the first instruction of the new program.

d. Error Conditions

This function returns an error to the caller if the memory requirements of the new program are invalid.

2. exit()

a. Parameters

The current process's UVECTOR, PROC, and TEXT are implied parameters.

b. Functional Description

This function is the system call used to terminate the calling process. It clears all signals, closes any open files, unlinks from the current TEXT, acquires a block on the swap device, copies the first 256 bytes of the UVECTOR to the block, and frees main memory space held by the process. In SUNIX and SSUNIX, the old memory area is freed by the new function "prfree". The new function "swfree" is used to free any swap space. In SSUNIX only, if a process has the SLOCK bit set in its PROC, the

function "shfree" is called to free any shared core assets that the process has been allocated.

c. Returned Values

This system call does not return to its caller. The next function invoked for this process is "wait", which completes the cleanup.

d. Error Conditions

This system call has no error conditions.

3. sbreak()

a. Parameters

The current process's UVECTOR and PROC are implied parameters. Because this function is a system call, an additional argument, the virtual address of the new end of the data, is found in the array u.u_ar0[] in the UVECTOR.

b. Functional Description

This function is the system call used to change the size of the calling process's data area. It calculates the new data size desired by the process and checks the validity of the process's new total memory requirements. In UNIX, "expand" is used to establish the new region. In UNIX and SSUNIX, this function attempts to do the work itself. It puts the new size in p_dsize in the current PROC. If the new size is smaller, it simply frees the excess storage. If the new size is larger, it attempts to acquire it. If this fails, "deswap" is called to acquire the space by swapping.

In all systems, the newly acquired space is cleared.

c. Returned Values

The values returned by this function are not checked.

d. Error Conditions

If the new storage requirement is not valid, the new space will not be acquired and the function returns. This will usually cause the process to terminate abnormally because of a memory management error.

G. text.c

1. xswap(p,ff,os), xswap(p,ff,ods,oss)

a. Parameters

In UNIX, this function is called with three arguments. In SUNIX and SSUNIX, it is called with four. The first parameter is a pointer to the proc of a process to be swapped out of memory. The second parameter is the memory free flag. In UNIX, the third parameter is the process image size in 64-byte blocks. In SUNIX and SSUNIX, the third and fourth parameters are the sizes of the process's data and stack segments in 64-byte blocks.

b. Functional Description

In UNIX, this function allocates swap space for the process and swaps it out. In SUNIX and SSUNIX, this function allocates space only for those processes that do

not already have it. In all systems, memory is freed if the memory free flag is set. This flag will not be set if this function was called by "newproc" to create a copy of the parent process. In SSUNIX, when this function is called to swap out a process with shared core, the memory free flag is not set if other memory-sharing processes remain in the shared region.

c. Returned Values

The values returned by this function are not checked.

d. Error Conditions

If swap space must be allocated, but none is available, the message "out of swap space" will be sent to the console and the system will crash. If an output error occurs during the swap operation, the message "swap error" will be sent to the console and the system will crash.

2. xalloc(ip)

a. Parameters

The parameter is a pointer to the inode of the text segment that is to be allocated or located. The current process's UVECTOR and PROC and all TEXTS are implied parameters.

b. Functional Description

This function establishes the shared text segment required by the current process. If the current

process does not require shared text, this function simply returns. If the process does require shared text, this function searches the array of TEXTS for a previously established TEXT for the requested segment. If one is found, its active-process use count is incremented. If the requested segment is in memory, the TEXTs in-memory count is incremented. If a TEXT has not been established, an unallocated TEXT is located and allocated to the text segment. Swap space is allocated for the text segment. The current process's address space is increased using "expand" to acquire space for the new text segment. The text segment is then read into memory and copied to the swap space allocated for it. The new memory space acquired for the text segment is freed using "expand" in UNIX and "prfree" in SUNIX and SSUNIX. The address of the TEXT is placed in p->textp in the current PROC and the process is swapped out of memory. When it returns to memory, the newly established text segment will return with it.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

If a TEXT must be established and one is not available, the message "out of text" will be sent to the console and the system will crash. If swap space must be allocated and none is available, the message "out of swap space" will be sent to the console and the system will

crash.

H. page.c

1. pralloc(pr)

a. Parameters

The parameter is a pointer to a PROC. The TEXT pointed to by the PROC is an implied parameter.

b. Functional Description

This function is present only in SUNIX and SSUNIX. It acquires memory space for the process's UVECTOR, data segment, stack segment, and, if necessary, shared text segment. Space for the text segment is acquired only if the text is not resident in main memory. If allocation fails, all memory space previously allocated is freed.

c. Returned Values

If all allocations are successful, the physical block number of the base of the UVECTOR is returned. If any allocation fails, a zero is returned.

d. Error Conditions

A return value of zero indicates an error to the caller.

2. prswap(rp)

a. Parameters

The first parameter is a pointer to a PROC. The TEXT pointed to by the PROC is an implied parameter.

b. Functional Description

This function is present only in SUNIX and SSUNIX. It swaps a process's UVECTOR, data segment, stack segment, and, if necessary, text segment into main memory.

c. Returned Values

The value returned by this function is not checked.

d. Error Conditions

If an input error occurs during the swap operation, the message "swap error" is sent to the console and the system will crash.

II. APPENDIX C: SYSTEM BENCHMARKS

A. BENCH 1, MONOPROGRAMMING

```
chdir /usr/sys
sh rpload
chdir ken
cc -O -c slp.c
cd ..
cd dmr
ed ipc.c </usr/bench/edcmd >/dev/null
chdir /usr/bench
cc -O rftest.c
bas tower<towerin>/dev/null
od  /usr/sys/conf/m45.s >/dev/null
cp /usr/unix /dev/null
chdir /bin
sum *>/dev/null
wait
chdir /usr/sys/ken
rm slp.o
chdir /usr/bench
rm a.out
```

B. BENCH 2, MULTIPROGRAMMING

```
chdir /usr/sys
sh rpload&
chdir ken
cc -O -c slp.c&
cd ..
cd dmr
ed ipc.c </usr/bench/edcmd >/dev/null&
chdir /usr/bench
cc -O rftest.c&
bas tower<towerin>/dev/null&
od  /usr/sys/conf/m45.s >/dev/null&
cp /usr/unix /dev/null&
chdir /bin
sum *>/dev/null&
wait
chdir /usr/sys/ken
rm slp.o
chdir /usr/bench
rm a.out
```

LIST OF REFERENCES

1. Digital Equipment Corporation, PDP-11/45 Processor Handbook, 1975.

2. Digital Equipment Corporation, PDP-11/34 Processor Handbook, 1976.

3. Digital Equipment Corporation, PDP 11 Peripherals Handbook, 1975.

4. Emery, H. W., The Development of a Partitioned Segmented Memory Manager for the UNIX Operating System, M. S. Thesis, Naval Postgraduate School, 1976.

5. Gray, P. E., The Design of a System Software/Hardware Interface for a Multiprocessor Graphics System, M. S. Thesis, Naval Postgraduate School, 1977.

6. Joy, R. E., Implementation of an Adaptive Scheduling Algorithm for the MUNIX Operating System, M. S. Thesis, Naval Postgraduate School, 1975.

7. Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," Communications of the ACM, v. 17, no. 7, p. 365-375, July, 1974.

8. Ritchie, D. M., The UNIX I/O System, Bell Telephone Laboratories, 1974.

9. Ritchie, D. M., C Reference Manual, Bell Telephone Laboratories, 1974.

10. Ritchie, D. M., UNIX Assembler Manual, Bell Telephone Laboratories, 1974.

11. Ritchie, D. M. and Thompson, K., UNIX Programmer's Manual, 6th ed., Bell Telephone Laboratories, 1975.

12. Visco, D. W., The Design of an Inter-Processor Support System for an Interactive Graphics Package, M. S. Thesis, Naval Postgraduate School, 1976.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4. Assistant Professor G. L. Barksdale, Code 52Ba Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5. Professor G. A. Rahe, Code 52Ra Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6. Computer Laboratory, Code 52ec Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
7. LT James M. O'Dell, USN USS La Moure County (LST 1194) FPO, New York 09501	1